A New Reachability Algorithm for Symmetric Multi-processor Architecture

Debashis Sahoo[1], Jawahar Jain[3], Subramanian Iyer[2], and David Dill[1]

[1] Stanford University, Stanford CA 94305, USA
[2] University of Texas at Austin, Austin, TX 78712, USA
[3] Fujitsu Labs of America

Abstract. Partitioned BDD-based algorithms have been proposed in the literature to solve the memory explosion problem in BDD-based verification. A naive parallelization of such algorithms is often ineffective as they have less parallelism. In this paper we present a novel parallel reachability approach that lead to a significantly faster verification on a Symmetric Multi-Processing architecture over the existing one-thread, one-CPU approaches. We identify the issues and bottlenecks in parallelizing BDD-based reachability algorithm. We show that in most cases our algorithm achieves good speedup compared to the existing sequential approaches.

1 Introduction

A common approach to formal verification of hardware is checking invariant properties of the design. Unbounded model checking [1, 2] of invariants is usually performed by doing a reachability analysis. This approach finds all the states reachable from the initial states and checks if the invariant is satisfied in these reachable states. However, exhausting the state space using the reachability approach is an intractable problem. Not surprisingly, such approaches suffer from the so-called *state explosion problem* for representing large state sets.

In practice, reachability analysis is typically done using *Reduced Ordered Binary Decision Diagrams* (OBDDs) [3, 4]. A more compact representation of boolean functions, *Partitioned-OBDDs* (POBDDs) [5] leads to further improvement in reachability analysis [6]. Various improvements to BDD data structures, variable ordering schemes, as well as the reachability algorithm itself have also been suggested to improve capturing the total reachable state space using reachability based verification. However, in practice the verification problem typically consumes far more resources than are typically available for even small sized problems of 100 state variables, and the gap between requirement and performance is continually growing.

The growing prevalence of, increasingly powerful, clustered high performance SMP (Symmetric Multi-Processing) machines appears to be an inevitable trend. However, it is not straightforward to devise a reachability algorithm to meaningfully use a very large number of processors.

D.A. Peled and Y.-K. Tsay (Eds.): ATVA 2005, LNCS 3707, pp. 26–38, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Given the above two trends, it is important to develop efficient parallel verification algorithms that can appropriately exploit the SMP architecture. Though the intractability of the problem will remain, the verification time can get reduced by a significant factor.

In this paper, we show that the naive parallelization of the POBDD-based reachability analysis doesn't have good parallelism. We present a novel parallel reachability approach that improves the parallelism. Our algorithm also improves the performance of sequential POBDD based approaches drastically in some cases. This is because, in sequential POBDD-based algorithms, the relative order in which the partitions are analyzed plays a critical role in the overall performance. Finding an optimal schedule is a very hard problem. Therefore, any heuristic to find a good schedule is likely to not perform well in all cases. In a few cases, the approach can get stuck in some difficult partition and, hence, many remaining states which otherwise could have been easily computed are not reached at all. Our algorithm clearly obviates this *scheduling problem* since it runs all partitions in parallel. Also, in a parallel shared-memory environment, using our techniques of *Early Communication* and *Partial Communication*, state space traversal in some partitions can continue even while remaining partitions are proving to be difficult.

We show that in most cases our algorithm performs much better than the corresponding sequential run using 8 processors. Using our approach, we can locate error states significantly faster than other BDD based methods. We can also show that our results are much better than the standard reachability algorithms in many passing cases as well. Finally, we show that our method is more robust than the standard sequential POBDD-based reachability algorithm as it is able to solve various easy reachability instances which prove to be problematic for current POBDD approaches.

2 Preliminaries

Reachability analysis is usually based on a breadth-first traversal of finite-state machines [4, 2]. The algorithm takes as inputs the set of initial states and a transition relation (TR) that relates the next states a system can reach from each current state. The set of reachable states is obtained by repeatedly performing image computations until a fixed point is reached [4, 2]. This is termed as the *Least Fixed Point* computation. Verification based on reachability can often be improved by the use of POBDDs [7, 6, 8]. Essentially, the POBDD based-reachability algorithm performs as many steps as possible of image computation within each partition i in a step of *least fixed point* within the partition. When no more images can be thus computed, it synchronizes between partitions by considering the transitions that originate in partition i and lead out from there. The term *Communication* refers to these cross-partition image computations that are followed by transferring the computed BDDs to other partitions. Notice that the POBDD-based reachability algorithm performs a BFS which is local to individual partitions, and then synchronizes to add states that result from transitions crossing over from one partition to another. We may charac-

terize this as a region-based BFS, where individual regions of the state space, *i.e,* the partitions, are traversed independently in a breadth first manner. We term the computation within individual partitions as a *local Least Fixed Point* computation or a local LFP computation in short.

Related Work

Several methods have been proposed to do parallel verification. Stern and Dill [9] parallelize an explicit model checker. In [10], parallelized BDDs are used for reachability analysis. Verification using parallel reachability analysis has been studied in [11, 12, 13]. A scalable parallel reachability analysis is presented in [12]. They perform distributed reachability using the classical BFS traversal of the state space in a parallel environment, using distributed memory. A different disjunctive partitioning approach based on iterative squaring is explored in [14]. A thread-based approach has been applied to Constraint-Based Verification in [15].

We implemented our algorithm as a multi-threaded program. We would like to compare our algorithm with other distributed approaches. However, at the time of submission of this paper, we didn't have an implementation of other distributed algorithms to compare with our approach. Therefore, we keep this as a future work.

3 Improving Parallelism in the Reachability Analysis

The reachability analysis involves construction of a TR and the actual reachability steps using the TR. We use the standard sequential approach of building the transition relation. We keep the parallelization of the construction of the transition relation as a future work. In this paper we parallelize the reachability algorithm using various heuristic improvement.

The POBDD-based algorithm given in [6] is naturally parallelizable. The local LFP computation of each partition combined with their *communication* can be processed in parallel. We have to wait for all the partitions to finish their local LFP computation and the *communication* to begin transferring the communicated states to the appropriate partition. However, empirically we find that this simple parallelization of the algorithm in [6] doesn't have much parallelism. This may be due to following reasons

High Variation of BDD Computations
The performance of the image computations inside each partition depend on the BDD variable order. We call a partition an *easy partition* if the BDDs inside the partition are compact and a hard partition otherwise. For a majority of circuits, the complexity of the BDD computations can have significant variations between different partitions. In such cases, all easy partitions wait for the hard partitions to finish their image computation, which reduces the parallelism significantly.

Depth of the local LFP computation
Another reason for the reduced parallelism may be because the depth of the local LFP computation can vary a lot between partitions. In this case the partition

with smaller depth finish faster whereas the partitions with larger depth take longer time. This results in many idle processors which reduces the parallelism.

In practice we find that a large number of partitions wait for a few hard partitions. To address this issue we use following heuristics[16] to improve the parallelism.

Early Communication: Communicate states to other partition after the least fixed point.
Partial Communication: Initiate a partial communication in an idle processor.

3.1 Early Communication

After a partition finishes its local LFP computation, we allow the partition to immediately communicate its states to the other partitions. Each partition accepts this communicated states asynchronously during their local LFP computation. This would enable the easy partitions to make progress with their subsequent local LFP computation without waiting for the hard partitions to finish. Therefore, the early communication from easy partitions to other easy partitions enables all such partitions to reach a fixed point. This is very difficult to achieve in sequential partitioned reachability analysis because such scheduling information is difficult to obtain.

If new states are *communicated* during early communication, then we restart the current image computation after adding these states. Such augmentation can make a harder image computation significantly easier in some cases. This may be because the states that would have been hard to compute in one partition can be more easily computed in another partition and then communicated to the first partition.

3.2 Partial Communication

Even after applying the above technique, we found that some partition that have completed the local LFP on their current states were waiting for other partitions to communicate some states, so that they can continue their local LFP computation. This case arises when all the easy partition finish their local LFP and need communication from a hard partition to make further progress. To improve parallelism, the active partition initiates a *communication* in an idle processor using a small subset of the state space of the hard partition. The *communication* introduces new states in the easy partitions. This enables easy partitions to make progress further with their collective least fixed point from the communicated states. Intuitively this tries to accelerate the activity among easy partitions. We found that communicating the full BDD to a different partition is very hard. Therefore, we find a small subset of state space that can be expressed with a compact BDD (High Density BDD[17]). This heuristic tries to keep all the processors busy there by improving the parallelism. Further, this heuristic can increase the number of early communication instances. Thus, the combined effect of the partial communication and early communication improves the parallelism significantly.

```
Parallel-Reachability(n, TR, InitStates) {
    Create n partitions for InitStates
    Run in parallel for each partition i{
        After every microsteps runs
                ImproveParallelism(i) {
            Get all the communicated states
            Calculate LeastFixedPoint(Rch) in partition i
            Compute cross-over states from i to all parts
        }
    } until (No new state is found in any partition);
}
ImproveParallelism(n: Partition Number) {
    check and add all the communicated states
        if new states are added
            restart current image computation
    request a waiting partition to initiate
        partial communication procedure
}
```

Fig. 1. Parallel Reachability Algorithm

3.3 Parallel Reachability Algorithm

We present our complete parallel POBDD-based reachability algorithm as shown in Figure 1 using the techniques discussed in last section.

We run the local LFP computation combined with the *Communication* in parallel. All computation inside a partition is managed by a dedicated processor. Each processor polls for the communicated states from the other processor. After every micro-step of the image computation, each processor calls a function *ImproveParallelism* that implements two heuristics for improving parallelism. The first heuristic is to do early communication. As a part of the first heuristic, the function checks whether other processors have communicated some states to the current partition. If it finds any processors, then it transfer all the communicated states from their corresponding partitions to the current partition. This simple check and update subroutine performed by each processor implements the early communication heuristic. The second heuristic is to do partial communication. As a part of this heuristic, every active processor checks for an idle thread. If an idle processor is found, then it gives a small subset of the state space from the current partition to the idle processor. The idle processor start a *Communication* from this subset of states to the partition associated with the idle processor.

3.4 Termination Condition

In our approach, each processor manages a partition. The processor goes back to idle state if no new states are communicated to the partition associated with that processor. One of the processor manages the global termination conditions. The processor asserts a global termination flag if all the processors are idle.

4 Engineering Issues

Our implementation of the POBDD-data structure and algorithms uses VIS-2.0 package. The VIS-2.0 package uses CUDD [18] for the BDD operations. We implemented our parallel reachability algorithm as a multi-threaded program in a symmetric multi-processing (SMP) architecture. SMP systems can be programmed using several different methods. In a multi-threaded approach, the program divides the work across the processors by spawning multiple light-weight threads, each executing on a different processor and performing part of the calculation. Since all threads share the same program space, there is no need for any explicit communication calls. However, designing a multi-threaded FV approach using BDDs poses significant challenges.

BDD Issues in Multi-threaded Reachability: The CUDD BDD package is designed for use in a non-thread based environment. Further, there are various optimization features in CUDD, that prevent it to function correctly in a multi-threaded environment. It uses many global variables, which needs to be synchronized in a multi-threaded environment. Nevertheless, fixing this problem enables the program to behave correctly provided each thread work on their respective BDD-managers. However, this leads to a non-deterministic behavior in the BDD-computation.

The CUDD package uses various memory based optimization to boost its performance. However, such optimizations behave non-deterministically in a multi-threaded environment. Therefore, the produced computation trace is often non-reproducible and the program becomes very difficult to debug. It also results in many orders of magnitude difference in run times. Thus, the program behavior is not predictable. However, deterministic behavior of the program is very important for the evaluation of its performance. We re-engineered all the relevant features in the CUDD package that leads to a non-deterministic behavior. This enables the BDD-package to be safe to run in a multi-threaded environment and makes the program more conveniently analyzable. However, this was surprisingly painful to implement.

In addition to the above, each thread needs to synchronize based on a deterministic measure before communicating to another thread. Otherwise, the program would behave non-deterministically because of the non-determinism in the thread scheduling. We synchronize the threads using a fixed count based on the number of BDD conjunction operations and the number of sift operations during variable reordering. Further, we find that the deterministic version of the program performs as good as the non-deterministic program as described in Section 5.2.

Performance Issues on SMP Machine: Further, the scheduling of the threads in an SMP machine, although improved significantly over the years, might not be optimal for our application. Each thread, in our case use separate BDD managers for carrying out various BDD operations. Therefore, if the system thread scheduler assigns the thread to a different processor, then the thread would loose all its cached data and the new processor would re-fetch all

the necessary data to carry out the BDD operations. Thus, assigning a thread to a new processor would incur unnecessary large overhead. However, a very simple scheduling strategy of assigning each thread to an exclusive processor would reduce the overhead generated by the heavy cache misses significantly. On the other hand, it is quite difficult to quantify the performance penalty due the non-optimality of scheduling threads.

Performance Issues on Uniprocessor Machine: Furthermore, the simulated parallel execution of the multi-threaded algorithm in a uniprocessor machine may perform better than other sequential algorithm because of the scheduling flexibility. However, the program may have large overhead due to the cache misses because of the frequent switching of threads in one processor. We find that reducing the frequency of switching of threads in a uniprocessor machine significantly improve the results. Moreover, a simulated sequential approach in an 8-CPU machine, where each thread can potentially use different processor cache improves the results further. We use explicit locks to run one thread at a time in the 8-CPU machine. We find that the performance in this simulated case is 2-6 times faster than the corresponding uniprocessor run. Thus, the uniprocessor performance is significantly penalized by the cache overhead. Therefore, we provide results from this simulated sequential approach in the 8-CPU machine in our final table to give a good overview of the parallelism achieved. However, the performance in any uniprocessor machine is much worse than the simulated sequential case in an 8-CPU machine.

5 Experimental Results

We run our experiments using default cluster size of 5000, lazy sift reordering, MLP image method on a 8-way SMP Linux machine based on Intel(R) Xeon(TM) MP CPU 2.20GHz and 8GB RAM. We run all the sequential algorithms on a Linux box with Intel(R) XEON(TM) CPU 2.20GHz and 2GB RAM. We report results only on a few VIS-verilog [19] and industrial circuits because of limited time. In keeping with the typical timeout limits set in our in-house verification tools, we set a timeout of 5000 seconds on all circuits. For sake of brevity, we present our results only on those circuits where VIS requires more than 100 seconds. Results are omitted for the circuits where all the methods timeout. We use 8 different partitions for all POBDD-based approaches. We select the partitioning variable using the method in [6]. We use same partitioning strategy for all partitioned approaches in order to perform a fair comparison.

5.1 Overview of Table

Table 1 shows our invariant check results on various public and industrial circuits. In Table 1, we separate the total reachability time into the transition relation construction time and the actual reachability time. We compare the actual reachability time taken by the following approaches: the standard approach of VIS, the simple partitioning approach and our parallel POBDD-based reachability algorithms. We compare the naive parallel approach with the successive

Table 1. Time (in sec) for Invariant Checking on a few VIS-verilog and Industrial Circuits using 8 CPUs

ckts	TR time	vis	seq pobdd	Parallel 8 CPUs (naive)	Parallel 8 CPUs (early comm)	(early comm + partial comm) **Parallel 8 CPUs**	(early comm + partial comm) Simulated Seq
(a) Industrial Circuits							
c1	36	371	T/O	T/O	T/O	**227**	286
c2	12	3346	1789	1564	93	**917**	917
c3	17	2540	T/O	T/O	T/O	**62**	228
c4	11	2236	2084	1174	161	**161**	509
(b) Few VIS-benchmark Circuits							
spprod	5	891	61	53	93	**440**	510
am2910	9	T/O	281	122	204	**356**	386
palu	3	273	4	9	8	**9**	9
s1269b-1	2	3635	T/O	T/O	59	**60**	72
s1269b-5	2	2287	T/O	T/O	55	**55**	67
blkjack-3	2	T/O	1213	470	340	**70**	98
(c) Simple Industrial Circuits							
d1	11	6	T/O	T/O	13	**13**	13
d2	15	10	11	13	45	**30**	39
d3	12	15	21	23	100	**100**	130
d4	8	11	T/O	T/O	39	**38**	60
d5	7	12	16	15	34	**37**	37

(T/O = Timeout of 5000 sec)

introduction of the two heuristics for communication – early communication and partial communication. The columns in the table are arranged in the same order. The first column is the circuit name, followed by transition relation construction time, *vis*, *seq*uential POBDDs, *naive* parallelization, the parallel approach with just early communication and finally with both techniques. The final column has two parts – *8 CPUs* and *Simulated Seq*, which report, respectively, the total reachability time in a parallel environment using 8 CPUs and the time in a simulated sequential approach in an 8-CPU machine. The simulated sequential approach is discussed in section 4. Note that many of the sequential results are better than standard POBDD-based reachability because of the partition and communication scheduling flexibility. The details of the processor utilization are presented in Section 5.3 using Gantt charts.

5.2 Efficiency Issues

Table 1 is composed of three different sections. Section (a) and (c), respectively shows the results on a few hard and easy industrial circuits. Section (b) shows the

Table 2. Time (in sec) for Invariant Checking on the Industrial Circuits using different redundancy value in a parallel and sequential framework

	redundancy [6]					
	0.3		0.5		0.7	
	Parallel	seq	Parallel	seq	Parallel	seq
c1	227	288	226	286	229	292
c2	73	386	917	917	2569	2570
c3	1492	1493	62	228	1407	T/O
c4	2967	2970	161	509	158	520
d1	26	28	13	13	92	138
d2	30	40	30	39	31	39
d3	53	67	100	130	102	133
d4	29	37	38	60	38	59
d5	13	13	37	37	37	38
s1269b-1	61	73	60	72	165	183
sp_prod	446	510	440	510	259	260

(T/O = Timeout of 5000 sec)

results on a few VIS-verilog benchmark circuits. As can be seen from the table, the resulting parallel run times with all the heuristics, *i.e*, the last column of the table have no timeouts. They are also clearly superior to classical partitioned-reachability. The proposed parallel approach will all heuristics, is also usually superior to the less sophisticated parallel techniques. The parallel approach with only early communication, i.e the 6th column in Table 1, often works well and have fewer timeouts compared to the naive parallel approach. Consider the circuit *blkjack-3*, which represents the best scenario, where the results improve with each successive addition of the heuristics. We find that the parallel approach is usually more robust than the sequential approaches. Note that the last column shows the results of simulated sequential approach in an 8-CPU machine to demonstrate the parallelism achieved. The corresponding uniprocessor results are 2-6 times worse than the simulated sequential approach. We find that the parallelism is very small and hope to improve it in a future work.

Scheduling is a Problem Even on Easy Functions: Consider the results of some properties from an industrial design whose OBDDs are fairly small as shown in Table 1 (c). The partitioned reachability for such cases gets harder. Both the standard sequential POBDD-based reachability and naive parallel reachability falls in the trap of an inefficient computation. An early communication often helps in this case, as can be seen from the table. However, both early communication and partial communication are needed to finish all the circuits. The reachability of small circuits using 8 partitions might contribute to some overhead in the partitioned reachability approaches.

Further, we will like to comment on the relative speedup of the multi-threaded 8-CPU approach over the simulated sequential approach. This speedup is not only proportional to the algorithm but also to the choice of partitioning variables.

Table 3. Time (in sec) for Invariant Checking on the Industrial Circuits using the non-deterministic and the deterministic program

ckts	Time in sec	
	non-det	det
(a) Industrial Circuits		
c1	T/O	**227**
c2	962	**917**
c3	809	**62**
c4	903	**161**
(b) Simple Industrial Circuits		
d1	13	**13**
d2	24	**30**
d3	84	**100**
d4	30	**38**
d5	13	**37**

(T/O = Timeout of 5000 sec)

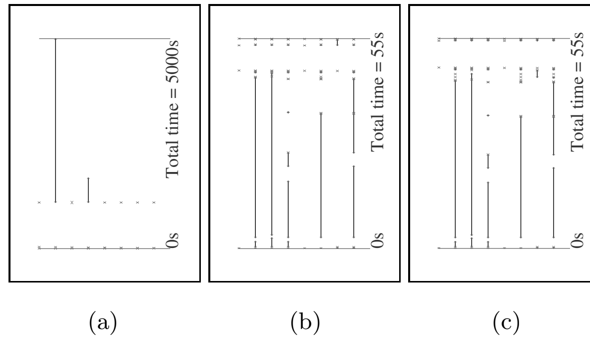
For the same algorithm, even though the *same* partitioning variables may be provided to both the approaches, depending on the splitting choices, the amount of parallelism that is generated can vary dramatically. For example, in Table 2 it can be seen that for almost half of the entries, by varying redundancy and balancedness, the two parameters that are calculated for evaluating partitioning variables, the amount of parallelism that is generated can vary dramatically. This points to the need for an approach which can dynamically evaluate different choices in deciding the partitioning variables. Such an idea is motivated by the strong results presented in Sahoo et al. [8], where it was shown the successful BDD decisions can be taken if we generate different short traces of reachability computation for each choice and then make the required decision.

Finally, we show that the deterministic version of our program doesn't loose the performance by a great margin to the non-deterministic version. Table 3 shows the results of Invariant checking on the industrial circuits using both the non-deterministic and the deterministic version of our program. As we can see from the table, the performance of non-deterministic program is very similar to the deterministic program in the simple circuits, i.e. Table 3 (b). However, the performance of the deterministic program is better than the non-deterministic version in the hard circuits in Table 3 (a). Therefore, we strongly prefer the deterministic version to the non-deterministic version.

5.3 Improving Parallelism

Consider the reachability analysis of *s1269b-5* from the VIS Verilog benchmark suite. As shown in Table 1 (b), we perform reachability analysis using 8 partitions, each of which runs in a separate thread.

Figure 2 shows the Gantt charts of three parallel reachability analysis on *s1269b-5* circuit. We use the three charts to show the effect of the two heuris-



(a) (b) (c)

Fig. 2. Parallel Reachability with successive addition of each heuristics

tics added successively to the reachability algorithm. Figure 2(a) shows Gantt chart of the naive parallel reachability. Figure 2(b) shows the Gantt chart of reachability analysis when early communication is allowed. Figure 2(b) shows the Gantt chart of reachability analysis when both early communication and partial communication are allowed. Each partition is represented by a vertical broken line. The filled segment represents the *cpu time* for the partition to perform a computation. At the end of each such stage, a small cross indicates the communication of states to other partitions. A break in the line indicates that the corresponding processor is idle. However, in a multi-threaded uniprocessor environment, the processor can immediately schedule another thread for execution. The total time is the reachability time on a multi-processor machine. As we can see from the figure, more gaps are being filled with the addition of each heuristic. This shows a clear trend of improved parallelism in each case.

6 Conclusion

Partitioning based state space traversal approaches where reachability on each partition is processed independently appear very suited for parallelization. However, we find that a naive parallelization of such algorithms is often ineffective. In this paper we discuss an algorithm suitable for parallel reachability on a symmetric multi-processing architecture. We show that in most cases our algorithm achieves good speedup in a multi-processor shared memory environment, compared to the corresponding sequential run. Further, the parallel algorithm is significantly faster than both the standard sequential reachability algorithm as well as the existing partitioned approaches especially when the property is erroneous. We have made the multi-threaded program behavior deterministic. We found that the performance of both the non-deterministic and the deterministic program is similar.

Our investigation, one of the first in the area of a parallel reachability algorithm exploiting SMP architecture reveals that there are significant areas of performance improvements. These include improving scheduling of threads on

various processors, selecting window functions that can potentially enhance parallelism, and communication strategies between threads to decrease number of idle CPUs.

Acknowledgments

The authors thank Fujitsu Laboratories of America, Inc for their gifts to support the research. Prof. Dill thanks the NSF for support via grants CCR-012-1403. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] Clarke, E., Emerson, E.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Proc. IBM Workshop on Logics of Programs. Volume 131 of Lecture Notes in Computer Science. (1981)
[2] McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers (1993)
[3] Bryant, R.: Graph-based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers **C-35** (1986) 677–691
[4] Coudert, O., Berthet, C., Madre, J.C.: Verification of sequential machines based on symbolic execution. In: Proc. of the Workshop on Automatic Verification Methods for Finite State Systems. (1989)
[5] Jain, J.: et. al., Functional Partitioning for Verification and Related Problems. Brown/MIT VLSI Conference (1992)
[6] Narayan, A.: et. al., Reachability Analysis Using Partitioned-ROBDDs. In: ICCAD. (1997) 388–393
[7] Iyer, S., Sahoo, D., Stangier, C., Narayan, A., Jain, J.: Improved symbolic Verification Using Partitioning Techniques. In: Proc. of CHARME 2003. Volume 2860 of Lecture Notes in Computer Science. (2003)
[8] Sahoo, D., Iyer, S.: et. al., A Partitioning Methodology for BDD-based Verification. In: FMCAD. (2004)
[9] Stern, U., Dill, D.L.: Parallelizing the murphy verifier. In: CAV. (1997)
[10] Stornetta, T., Brewer, F.: Implementation of an efficient parallel BDD package. In: DAC. (1996) 641–644
[11] Garavel, H., Mateescu, R., Smarandache, I.: Parallel state space construction for model-checking. In: SPIN workshop on Model checking of software, Springer-Verlag New York, Inc. (2001) 217–234
[12] Heyman, T., Geist, D., Grumberg, O., Schuster, A.: Achieving scalability in parallel reachability analysis of very large circuits. In: CAV. (2000)
[13] Yang, B., O'Hallaron, D.R.: Parallel breadth-first bdd construction. In: symposium on Principles and practice of parallel programming, ACM Press (1997) 145–156
[14] Cabodi, G., Camurati, P., Lavagno, L., Quer, S.: Disjunctive partitioning and partial iterative squaring: An effective approach for symbolic traversal of large circuits. In: DAC. (1997) 728–733
[15] Pixley, C., Havlicek, J.: A verification synergy: Constraint-based verification. In: Electronic Design Processes. (2003)

[16] Sahoo, D., Jain, J., Iyer, S.K., Dill, D.L., Emerson, E.A.: Multi-threaded reachability. In: To appear In DAC. (2005)
[17] Ravi, K., Somenzi, F.: High-density reachability analysis. In: ICCAD. (1995) 154–158
[18] Somenzi, F.: CUDD: CU Decision Diagram Package ftp://vlsi.colorado.edu/pub (2001)
[19] VIS: Verilog Benchmarks http://vlsi.colorado.edu/˜ vis/ (2001)